NASA Contractor Report 181825

ICASE REPORT NO. 89-20

ICASE

COMPILING HIGH LEVEL CONSTRUCTS TO DISTRIBUTED MEMORY ARCHITECTURES

Piyush Mehrotra

John Van Rosendale

Contract No. NAS1-18605
March 1989

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Recently, ICASE has begun differentiating between reports with a mathematical or applied science theme and reports whose main emphasis is some aspect of computer science by producing the computer science reports with a yellow cover. The blue cover reports will now emphasize mathematical research. In all other aspects the reports will remain the same; in particular, they will continue to be submitted to the appropriate journals or conferences for formal publication.

Compiling High Level Constructs to Distributed Memory Architectures[†]

Piyush Mehrotra[‡], *John Van Rosendale*

Institute for Computer Applications in Science and Engineering

Abstract

Current languages for nonshared memory architectures provide a relatively low-level programming environment. In this paper we describe a set of language primitives which allow the programmer to express data-parallel algorithms at a higher level, while also permitting control over those aspects of the program critical to performance, such as load balance and data distribution. Given such a program specification, the compiler automatically generates a distributed program containing send and receive constructs to perform interprocess communication.

[†] This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-18605, Institute of Defense Analysis under contract IDA 10-00008, and Office of Naval Research under contract ONR N00014-88-M-0108 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

[‡] On leave from Dept. of Computer Science, Purdue University, West Lafayette, IN 47906.

1. Introduction

Nonshared memory architectures are currently programmed using message-passing languages, such as CSP[2] and Occam[6], in which the programmer defines a system of interacting "tasks" or "processes," which communicate through exchange of messages. These languages allow the user to fully control and exploit the underlying hardware, and are well suited to some classes of algorithms, such as game tree searching and discrete event simulation, where the algorithm decomposes naturally into a system of cooperating processes. However, for algorithms relying on synchronous manipulation of distributed data structures, as is typical in numerical computation, such languages have proven awkward. The problem is that the "abstractions" in which the programmer tends to think, for example, distributed arrays, are not well represented in the language.

In our approach, data parallel algorithms are specified as parallel loops acting on distributed data structures. The distribution of these data structures and the allocation of work to processors are separately specified. The compiler then maps this high-level specification into an interacting system of tasks, which communicate via message-passing.

The goal here is to allow the user to specify the algorithm at the highest possible level. At the same time, we wish to make the user explicitly aware of data distribution, load balancing, and communication costs, since these issues critically effect performance on nonshared memory architectures. The many small details involved in message exchange and synchronization are relegated to the compiler. With these choices, the programmer is free to focus on high-level algorithm design and performance issues, while relegating the minor but complex details of interprocessor communication to the compiler and run-time environment.

2. Language Primitives

The goal of our approach is to allow the programmers to treat distributed data structures as single objects. In our approach, the programmer must specify three things: a) the processor topology on which the program is to be executed, b) the distribution of the data structures across these processors, and c) the parallel loops and where they are to be executed. The following subsections describe each of these kinds of specification.

Processor Arrays

The first thing that needs to be specified is a "processor array." This is an array of physical processors across which the data structures will be distributed, and on which the algorithm will execute. Such a specification has the form:

```
int    np in 1 .. 8
procs  P[np, np]
```

These statements allocate a square array P of np^2 processors, where np is an integer constant between 1 and 8, dynamically chosen by the run-time system.

This construct provides a "real estate agent," as suggested by C. Seitz. Allowing the size of the processor array to be dynamically chosen is important here, since it provides portability, and avoids dead-lock in case a smaller than expected number of processors is available. The basic assumption here is that the underlying architecture can support multi-dimensional arrays of physical processor, an assumption natural for hypercubes and mesh connected architectures.

Data Distribution Primitives

Given a processor array, the programmer must specify the distribution of data structures across that processor array. The current version of our system supports only distributed arrays; other distributed data structures will be allowed in future versions.

Array distributions are specified by a *distribution clause* in their declaration. This clause specifies a sequence of distribution patterns, one for each dimension of the array. Scalar variables and arrays without a distribution clause are simply replicated, with one copy assigned to each of the processors in the processor array.

Each dimension of a data array can be distributed across the processors in one of several patterns, or can be left undistributed. The currently supported distribution patterns are **block** and **cyclic**. With a **block** distribution, each processor receives a contiguous block of elements of the array. Conversely, with a **cyclic** distribution, the array elements are distributed in a round-robin fashion across the processors. The number of dimensions of an array that are distributed must match the number of dimensions of the underlying processor array. Hyphens are used to indicate dimensions of data arrays which are not distributed.

Forall Loops

Operations on distributed data structures are specified by **forall** loops. The **forall** loop here is similar to that in BLAZE [5]. The example below shows a loop which performs 63 loop invocations, shifting the values in the array *A* one space to the left.

```
forall i in 1 : 63 on P(A[i]) loop
   . . .
   A[i] := A[i+1]
   . . .
end
```

The semantics here are "copy-in copy-out," in the sense that the values on the right hand side of the assignment are the old values in array *A*, before being modified by the loop. Thus the array *A* is effectively, "copied into" each invocation of the **forall** loop, and then the changes are "copied out."

In addition to the range specification in the header of the **forall**, there is also an **on** clause. This clause specifies the processor on which each loop invocation is to be executed. In the above program fragment, the **on** clause causes the *i*th loop invocation to be executed on the processor owning the *i*th element of the array *A*.

Given these primitives, a programmer can specify a data parallel algorithm at a high level, while still retaining control over those details critical to performance. As an example, the code fragment in Figure 1. performs a "smoothing" iteration on an array *X*, in which the new value of each element of *X* is an average of the values of its four neighbors. Note that the body of the **forall** loop is independent of the distribution of the array *X*, and of the processor array *P*. Thus a variety of distribution patterns could easily be tried by trivial modification of this program.

3. Structure of the Generated Code

Using the language primitives described in the last section, the user can provide a high level specification of the parallel algorithm. The compiler parses and analyzes the source code to produce a set of concurrently executing processes. The generated code runs in what has been termed the SPMD (Single Program Multiple Data) mode. That is, the same process code is down loaded onto each of the processors of the target architecture. The processes then execute asynchronously, interacting with each other via message-passing.

There are two major issues in restructuring the source code for parallel execution. First, the **forall** loops have to be partitioned among the processes, as specified by the **on** clauses in the loop header. Second, all remote accesses have to be "compiled" into message passing communication.

The first issue is straight forward. In the literature on restructuring compilers it is known as *strip-mining* or *loop-chunking*, and is a standard technique [1, 7]. The second issue is more subtle. All references to distributed data structures have to be analyzed, to identify potentially nonlocal accesses, and then converted into appropriate message passing

```
int   np  in  1 .. 4
procs  P[np, np]

real  X[0..N+1, 0..N+1]  dist  [block,block]

for k in 1 : 50 loop

   forall (i,j) in (1 : N, 1 : N) on P(X[i, j]) loop

      X[i,j] := 0.25*(X[i+1,j] + X[i-1,j] + X[i,j+1] + X[i,j-1])

   end
end
```

Figure 1: *Smoothing Iteration*

communication. The rest of this paper will focus on this second issue.

4. Induced Communication

Whether an array reference is local or not depends on the array distribution pattern, on the **on** clause in the **forall** loop header, and on the way the array elements are accessed. In some cases, the compiler has enough information to specify exactly where data values should be "mailed" in order that all loop invocations have the information they need before their

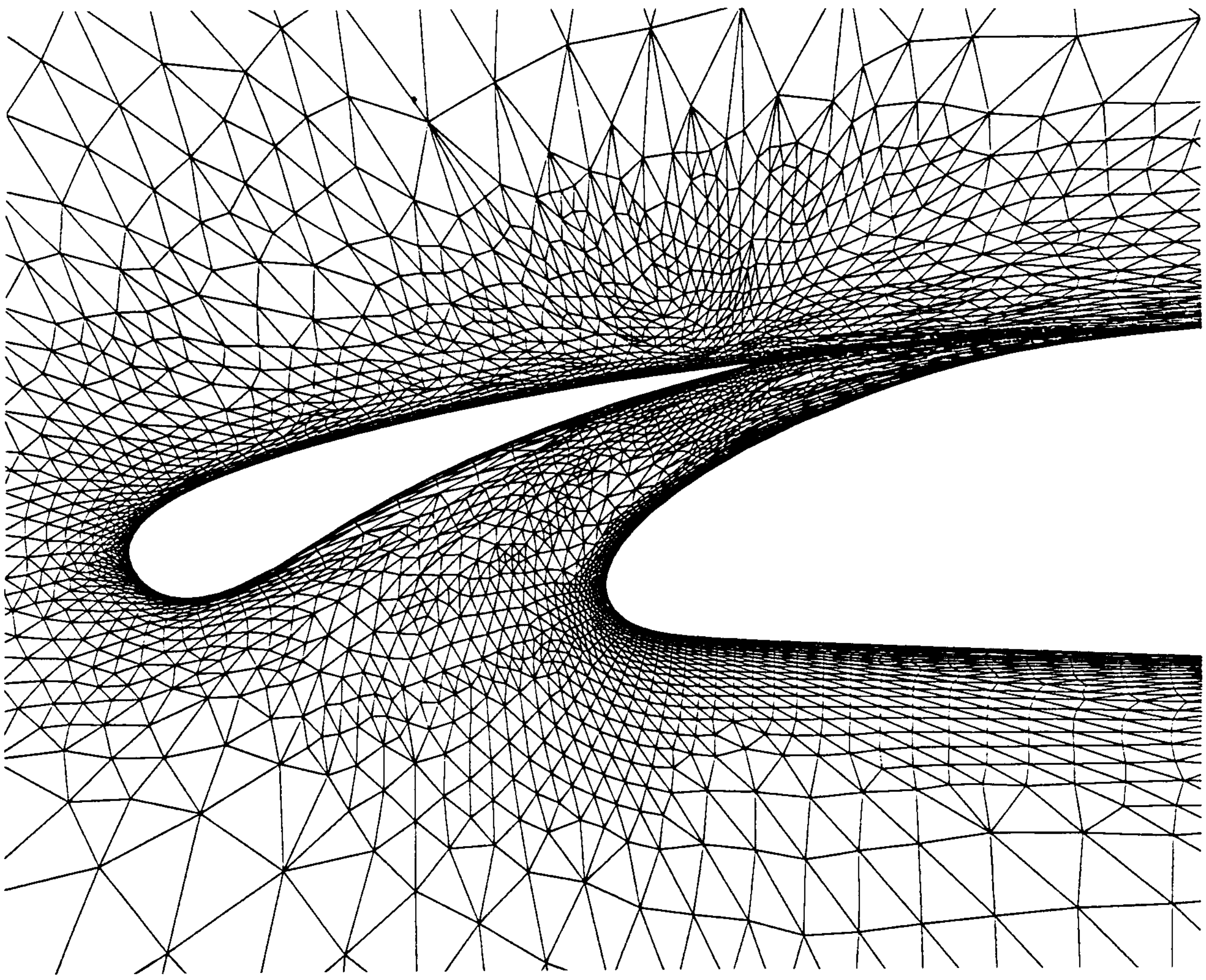

Figure 2: *Irregular Triangular Grid*

execution begins. This is the case for the smoothing iteration given in Figure 1. For each array reference, the compiler can determine which process owns the array element and which processes will need it in their loop invocations. It can then generate "optimal" message passing code.

In many cases the compiler does not have enough information to determine, at compile time, where to mail data in order to allow the **forall** loop invocations to execute without access to nonlocal data. This situation arises, for example, with irregular grids, such as that shown in Figure 2. Here the array accesses depend on an indexing function computed at run-time. In such situations, the compiler has to generate "fetches" to retrieve nonlocal data. The process requiring the data will send a request message to the process owning the data, which will mail the requested value back to the requesting process. The run-time code needed to carry this out is relatively complex, but fortunately can be done in an efficient manner by pulling all "fetches" out of the **forall** loop itself.

Consider the problem of performing a "smoothing" iteration on an irregular grid. Figure 2. shows an irregular grid for aerodynamics calculations, generated by D. Mavripilis [4]. For this grid, the smoothing iteration would be more complex, as shown in Figure 3.

Here, the values at the *N* nodes in the grid are represented as a one-dimensional array of values, *value*. Each node has a maximum of *MAX_NBRS* neighbors. The number of neighbors of each mesh point is stored in the vector *n_nbrs*, and the indices of these neighbors are stored in the two-dimensional array *nbrs*.

```
real value[N],    w[N, MAX_NBRS]     dist ...
int  n_nbrs[N],  nbrs[N, MAX_NBRS]  dist ...

nbrs := ...              % generate grid

while not done loop

    forall i in 1 : N on P(value[i]) loop

        for k in 1 : n_nbrs[i] loop

            value[i] := value[i] + w[i, k]*value[ nbrs[i, k] ]

        end
    end
end
```

Figure 3: *Smoothing iteration on irregular grid*

We suppose that the grid is generated on the fly by some algorithm. The values in the array *nbrs* are set at run-time, as indicated. The distributions of the arrays are not shown. Proper distribution of the arrays in this case raises load balancing issues outside the scope of this paper. The **forall** loop ranges over the mesh points in the grid. The new value at each mesh point is computed by the inner **for** loop, and is a weighted sum of the values at the mesh point's immediate neighbors.

The important point here is that in the **forall** loop, the elements of the vector *value* are indexed by the array *nbrs*. Thus the compiler cannot determine which elements will be accessed, and has to generate remote "fetches". A high-level pseudo-code version of the generated code is presented in Figure 4. The first time through the loop, each process determines which nonlocal values it needs, and sends a "fetch request" to the owning process of each nonlocal value it needs. While waiting for its requests to be answered, the process services requests that it receives from other processes. When it has received all its required data, the process increments a global counter signaling that it is done. It then continues servicing fetch requests from other processes until all processes have signaled receipt of their fetch requests. The global synchronization is required, since a priori knowledge of the number of fetch request each process will receive is not available.

During this first time through the loop, each process can keep track of the fetch requests it fulfills. This information can then be used in subsequent trips through the loop, to directly send the data where it is needed, without performing any fetchs. Thus the overhead of the fetch requests and the global synchronization is incurred only the first time through the loop. In general, a large number of smoothing iterations are required, so the overhead of servicing these fetch requests the first time through the forall loop should be relatively minor.

More Complex Examples

In the irregular grids example presented above, the run-time system can easily determine the remote data needed by each process, before loop execution begins. This is possible, since neither of the arrays *n_nbrs* and *nbrs* is modified in the **forall** loop body. Thus it is easy for the compiler to analyze and restructure the program so that all nonlocal accesses are done ahead of the loop.

In more complex situations, this may not be possible. Consider the case where the locality of each reference is intimately tied to the computation within the **forall** loop. In such situations, the fetches of remote data cannot be extracted from the body and must remain embedded within it. To handle these cases, one would have to set up the "threads" on each processor to service remote requests, in effect simulating shared memory on a nonshared memory architecture. The performance penalty in doing this would be severe.

Our current system handles cases like the irregular grid example here, and somewhat more complexes cases as well. Its main limitation is that the only control constructs allowed in **forall** loops are branching constructs, and **for** loops whose limits are set outside the forall loop. In particular, **while** loops are not allowed inside **forall** loops. Our goal at the moment is to demonstrate the efficacy of this approach in cases where we can guarantee good performance. Extending this approach to more difficult case, such as that of **while** loops

```
On each processor:

    - first time through while loop

        • communication phase
            - send remote fetch requests for all remote
              values needed on this processor
            - service fetch requests for other processors
              and receive requested data
            - continue to service (potential) fetch requests
                until all processors have received their requested data

        • computation phase
            - execute "strip-mined" loop

    - subsequent times through while loop

        • communication phase
            - send data needed by other processors
            - receive needed data from other processor

        • computation phase
            - execute "strip-mined" loop
```

Figure 4: *Code generated for irregular grids example*

within **forall** loops, is a subject for future research.

5. Conclusion

In this paper, we have presented a set of language primitives for nonshared memory architectures. These primitives allow specification of programs at a higher level than is possible with current message-passing languages. The programmer must still explicitly manage data distribution and load balancing, since these issues are critical to performance and cannot be automated with current compiler technology. However, the minor details of process management and of interprocessor communication and synchronization are relegated to the compiler, greatly reducing the burden on the programmer.

One of the principal advantages of this approach is that it allows the algorithm to be designed and specified in a distribution independent manner. Specifying the algorithm and data distribution separately simplifies programming, enhances portability, and permits easy "tuning" of programs, by allowing experimentation with a variety of data distributions and load balancing strategies, through minor changes in the program.

A preliminary version of these languages primitives is currently under development at Purdue University. Though we do not yet have performance statistics, in simple cases, such as those considered in this paper, the message passing code our system produces is virtually identical to that produced by experienced programmers, so will achieve the same performance.

References

1. Allen, J. R., "*Dependence Analysis for Subscripted Variables and Its Application to Program Transformations,*" PhD Thesis, Rice University, Houston TX, Apr., 1983.
2. Hoare C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, **21**(8), pp 666-677, Aug., 1978.
3. Koelbel, C., P. Mehrotra, and J. Van Rosendale, "Semi-automatic Process Partitioning for Parallel Computation," *International Journal of Parallel Programming*, **16**(5), pp 366-382, 1987.
4. Mavripilis, D. J., "*Adaptive Mesh Generation for Viscous Flows using Delaunay Triangulation,*" ICASE Report No. 88-47, (submitted to J. Comp. Physics), 1988.
5. Mehrotra, P. and J. Van Rosendale, "The BLAZE Language: A Parallel Language For Scientific Programs," *Parallel Computing*, **5**(3), pp. 339-361, Nov., 1987.
6. Pountain, D. "*A Tutorial Introduction to Occam Programming,*" Inmos, Colorado Springs, Co., 1986.
7. Wolfe, M. J., "*Optimizing Supercompilers for Supercomputers,*" PhD Thesis, University of Illinois, Urbana, IL, Oct., 1982.

National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-181825 ICASE Report No. 89-20	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle COMPILING HIGH LEVEL CONSTRUCTS TO DISTRIBUTED MEMORY ARCHITECTURES		5. Report Date March 1989
		6. Performing Organization Code
7. Author(s) Piyush Mehrotra John Van Rosendale		8. Performing Organization Report No. 89-20
		10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18605
		13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Proceedings of Hypercube
Multiprocessors 1989

Final Report

16. Abstract

Current languages for nonshared memory architectures provide a relative low-level programming environment. In this paper we describe a set of primitives which allow the programmer to express data-parallel algorithms at a higher level, while also permitting control over those aspects of the program critical to performance, such as load balance and data distribution. Given such a program specification, the compiler automatically generates a distributed program containing send and receive constructs to perform interprocess communication.

17. Key Words (Suggested by Author(s)) programming languages, nonshared memory architectures	18. Distribution Statement 61 - Computer Programming Software Unclassified - Unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 12	22. Price A03